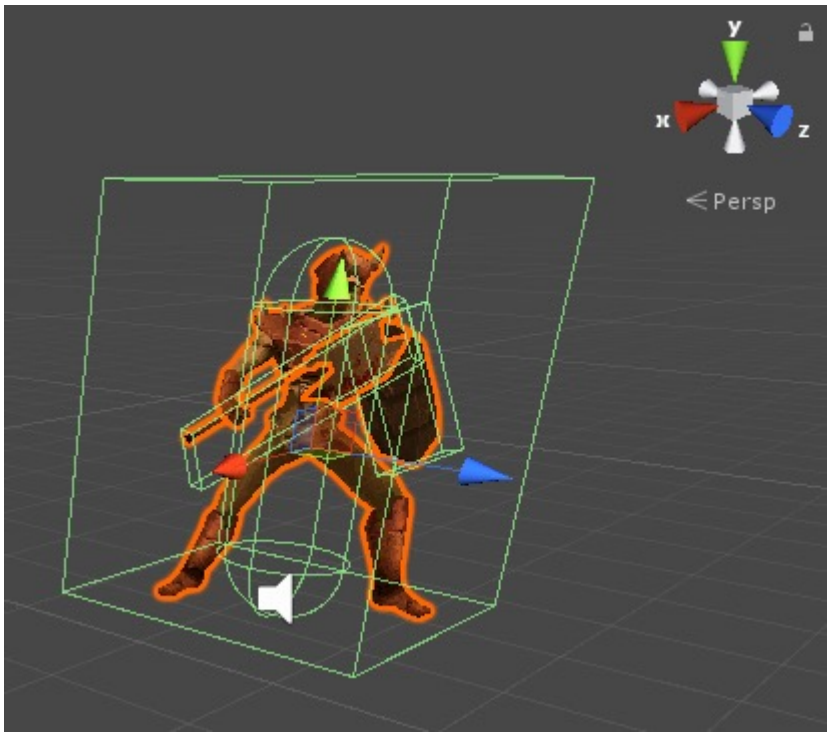
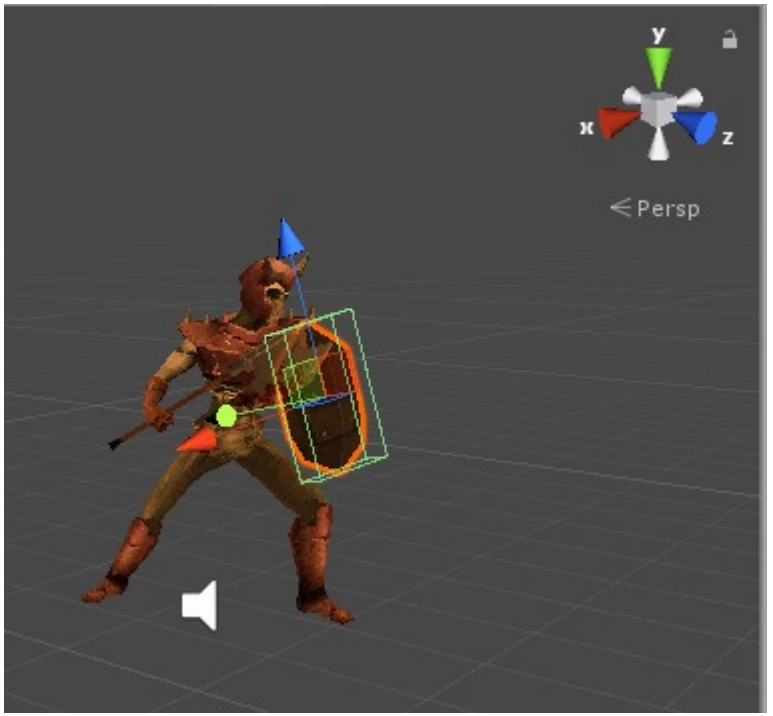
CT3536 (Games Programming) Section 3

- Coordinate systems: Global, Local, Nested, Screen-space, Viewport-space, World-space
- Coroutines
- Object interactions with Colliders and Triggers
- Layers
- Lab 3: Asteroid assault on Mars and its moons!

Coordinate Systems

- The most important distinction is between the Global (or World) coordinate system, and the Local coordinate system of each GameObject





Coordinate Systems

- Hence, to rotate the camera around the y-axis as perceived by the camera, (i.e. the y-axis of its Local Coordinate system) we can use:

```
camera.transform.RotateAround(Vector3.zero,
camera.transform.up, 50f * Time.deltaTime);
```

- Or, to rotate it around the Global y-axis:

```
camera.transform.RotateAround(Vector3.zero,
Vector3.up, 50f * Time.deltaTime);
```

- (The first Vector3 argument defines the world point around which to rotate)

Coordinate Systems

- Some methods of the Transform class, for translating from the local space of a transform, to global space:

```
public Vector3 TransformPoint(Vector3 position);
public Vector3 TransformDirection(Vector3 direction);
public Vector3 TransformVector(Vector3 vector);
```

- And for translating in the other direction (world – to – local) :

```
public Vector3 InverseTransformVector(Vector3 vector);
```

etc.

Coordinate Systems

- Example: to find the world-coordinate of a point which is 10 units 'in front of' a spaceship (in terms of its own direction facing, and assuming this code is located inside a script attached to the spaceship):

```
Vector3 pt = transform.TransformPoint(new Vector3(0f, 0f, 10f));
```

- Another way of doing the same thing would be:

```
Vector3 pt = transform.position + 10f * transform.forward;
```

- Example: accelerate a spaceship forwards (assuming it has a Rigidbody and we're using physics, and we're doing this in a FixedUpdate() method on one of its scripts):

```
Rigidbody rigid = GetComponent<Rigidbody>();
rigid.AddForce(transform.forward*200f*Time.fixedDeltaTime);
```

Coordinate Systems

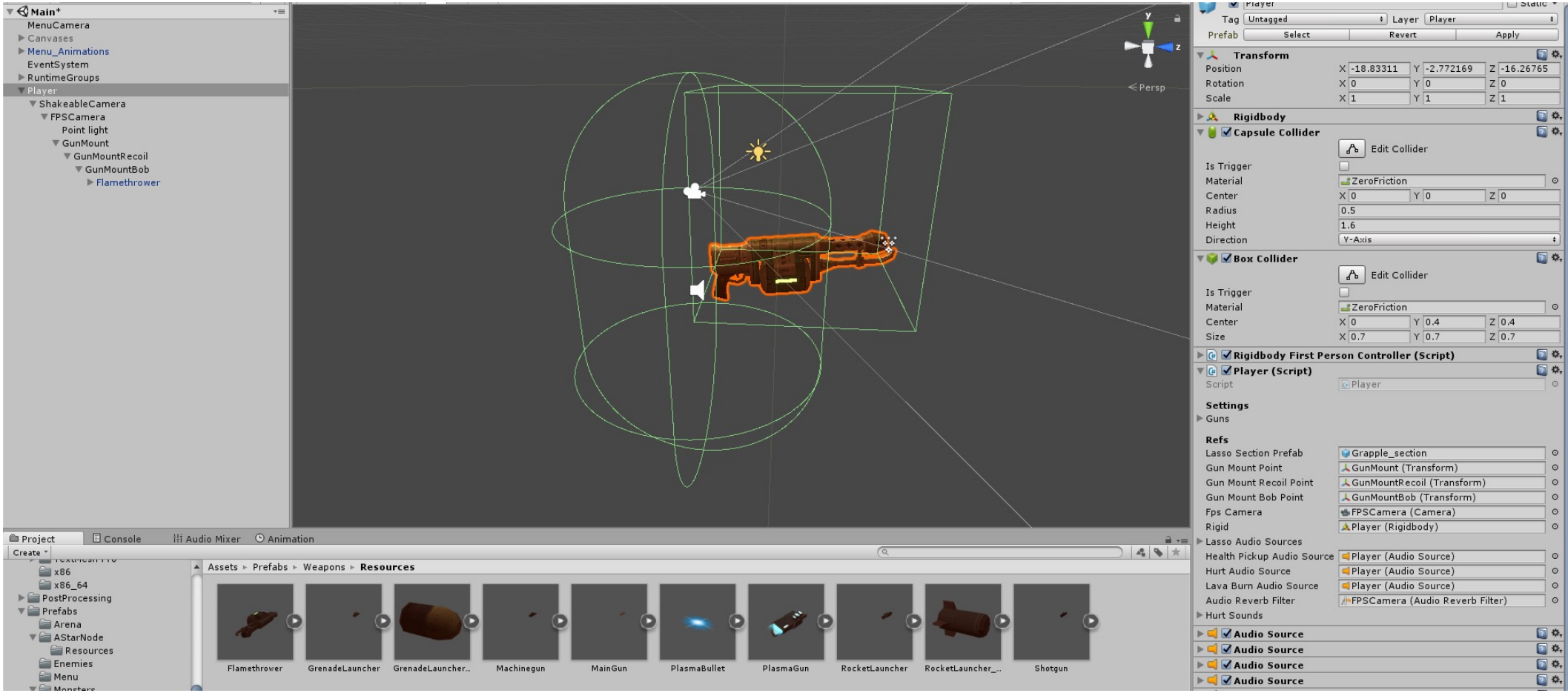
- More examples: get the direction and distance between two GameObjects:

```
GameObject go1, go2; // it's assumed that these are not nulls!

Vector3 difference, direction;
difference = go2.transform.position – go1.transform.position;
direction = difference.normalized;
float distance = difference.magnitude;
```

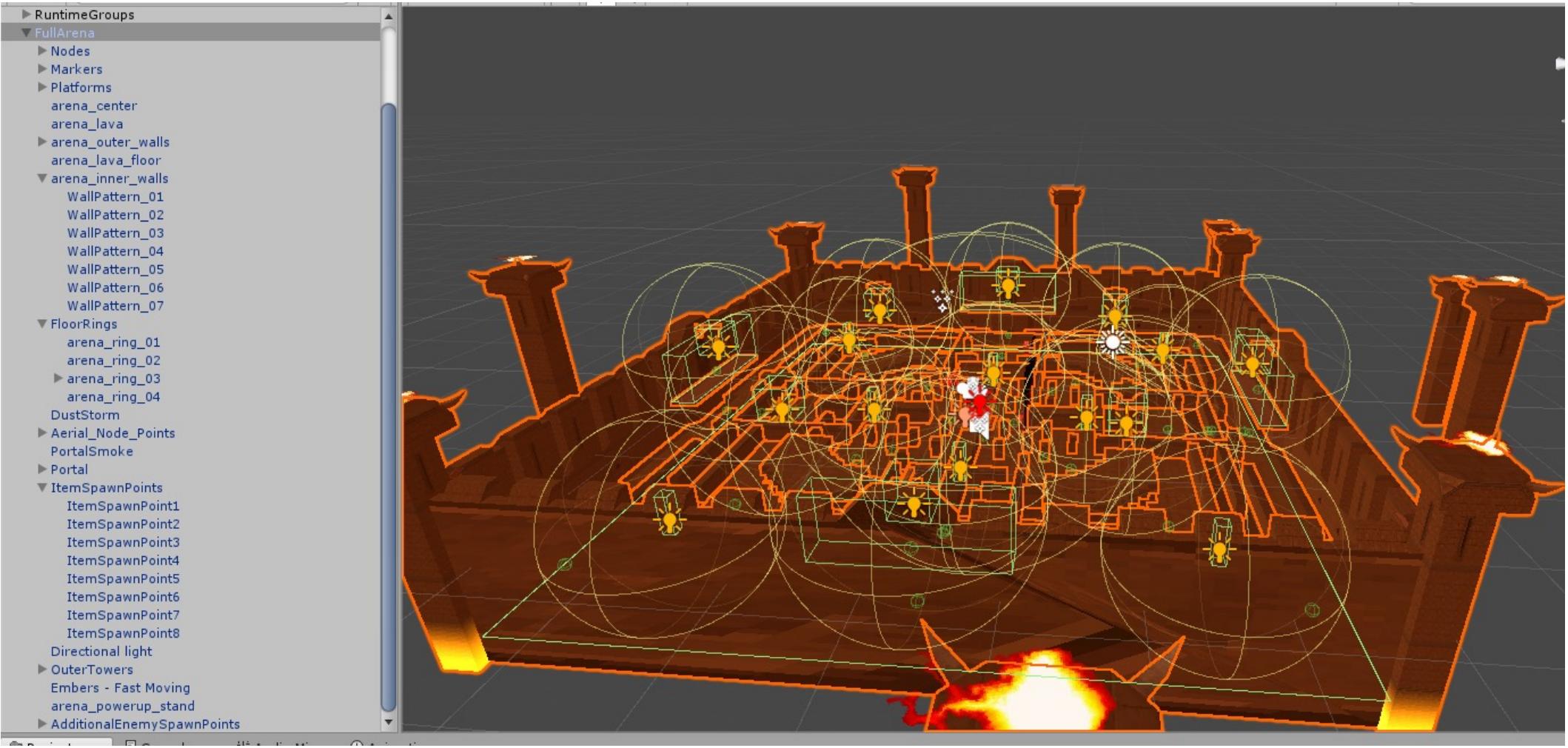
- Basically, you can't go far as a game programmer without being comfortable manipulating vectors
- But it really isn't difficult

Nested Coordinate Systems: The Demon Pit Player prefab



- The various nested Game Objects make it easy to mount things in the correct position - e.g. camera, light, currently equipped gun
- Several of the nested objects are related purely to camera and gun animations related to screen-shake, head movement while running, gun movement while running or shooting

A Complex Nested Prefab: The Demon Pit Arena Prefab



- The arena is constructed from many separate parts, including 'inner walls' and 'floor rings' which can change position (vertically) during the game, to create a changing arena layout
- By putting the nested objects into sensible groups, I can allow other team members to add/remove objects without needing to access any code (see next slide)

Iterating Nested Objects, using: foreach (Transform t in Transform)

```
public class Arena : MonoBehaviour {

  // inspector settings
  public Transform innerWallsGroup;
  public Vector3 wallsPos;
  //

  private List<GameObject> innerWalls = new List<GameObject>();
  private bool[] wallIsRaised = null;

  void Start() {

    foreach (Transform t in innerWallsGroup) {
      innerWalls.Add(t.gameObject);
    }

    wallIsRaised = new bool[innerWalls.Count];

    for (int i=0; i<innerWalls.Count; i++) {
      innerWalls[i].transform.position = wallsPos;
      innerWalls[i].SetActive(false);
      wallIsRaised[i] = false;
    }
  }
}
```

Screen-space, Viewport-space, World-space

- https://docs.unity3d.com/ScriptReference/Camera.html
- The Camera class provides methods for translating between three different 'spaces' – i.e. different ways of mapping the positions of things. Each of these is stored as a **Vector3** (which is a struct)
- A screen space point is defined in pixels. The bottom-left of the screen is (0,0); the right-top is (Camera.pixelWidth, Camera.pixelHeight).
  - If using a Vector3 (rather than Vector2), then the z position is in *world units* in front of the Camera, rather than in screen pixels.
  - Appropriate for e.g. mouse position
- Also see the Screen class: https://docs.unity3d.com/ScriptReference/Screen.html
- A viewport space point is normalized and relative to the Camera. The bottom-left of the Camera is (0,0); the top-right is (1,1).
  - Again, the z position (if any) is in world units in front of the Camera.
  - Appropriate for e.g. positioning GUI elements (independent of actual pixel resolution of the screen)
- A world space point is defined in global coordinates (for example, Transform.position).
  - Appropriate for e.g. GameObjects in the game world
- GameObjects which are nested in the Hierarchy have both Transform.position and Transform.localPosition; Transform.rotation and Transform.localRotation, etc.

Typical Space-Translation Operations

- Where is the mouse in world coordinates?
- E.g. to see if a GameObject is under the mouse

```
Vector3 mousePosInWorld =
      Camera.main.ScreenToWorldPoint(Input.mousePosition);
```

- Note that Input.mousePosition gives a Vector3 where the z component is 0
- The screen is 2D, of course. The z component of the Vector3 supplied to ScreenToWorldPoint is a world-coordinate distance into the world

- Where is a GameObject in screen-pixel coordinates?
- E.g. to position a GUI item such as a healthbar above it

```
GameObject targ; // assumed not to be null
Vector3 screenPos =
      Camera.main.WorldToScreenPoint(targ.transform.position);
```

Example

- The 'final enemy indicator' in DemonPit
- Here we find the 3D world position that's just above the enemy's head, and convert this to a viewport position when viewed through the player's camera
- Finally, this viewport position is converted to a pixel position as required by the GUI sprite

```
public class MonsterManager : MonoBehaviour {

void Update () {
   if (GameManager.gameState==GameStates.Playing) {
      if (numMonstersAlive==1) {
         // put the final enemy indicator (GUI object) above the final monster
         GameObject go = GUIManager.instance.finalEnemyIndicator;
         Monster m = allActiveMonsters[0];
         Bounds b = m.mycollider.bounds;
         Vector3 pos = new Vector3(b.center.x, b.max.y + 1f, b.center.z);
         Vector3 viewPos = Player.myPlayer.fpsCamera.WorldToViewportPoint(pos);
         if (viewPos.x<0f || viewPos.x>1f || viewPos.y<0f || viewPos.y>1f || viewPos.z<0f)
            go.SetActive(false);
         else {
            go.SetActive(true);
            go.transform.position = new Vector2(viewPos.x*Screen.width, viewPos.y*Screen.height);
         }
      }
   }
}
```

Object interactions with Colliders and Triggers

- Objects which have Colliders and RigidBodys will physically respond to impacts (i.e. they'll bounce off each other)
- Objects which have Triggers (but not colliders) will **not** physically interact

- We already know how to impart linear velocity and torque to a Rigidbody
- We already know about some of the settings that Rigidbodys have
- We will look into Unity's Physics engine in more detail later (section 4 of the module).

Object interactions with Colliders and Triggers

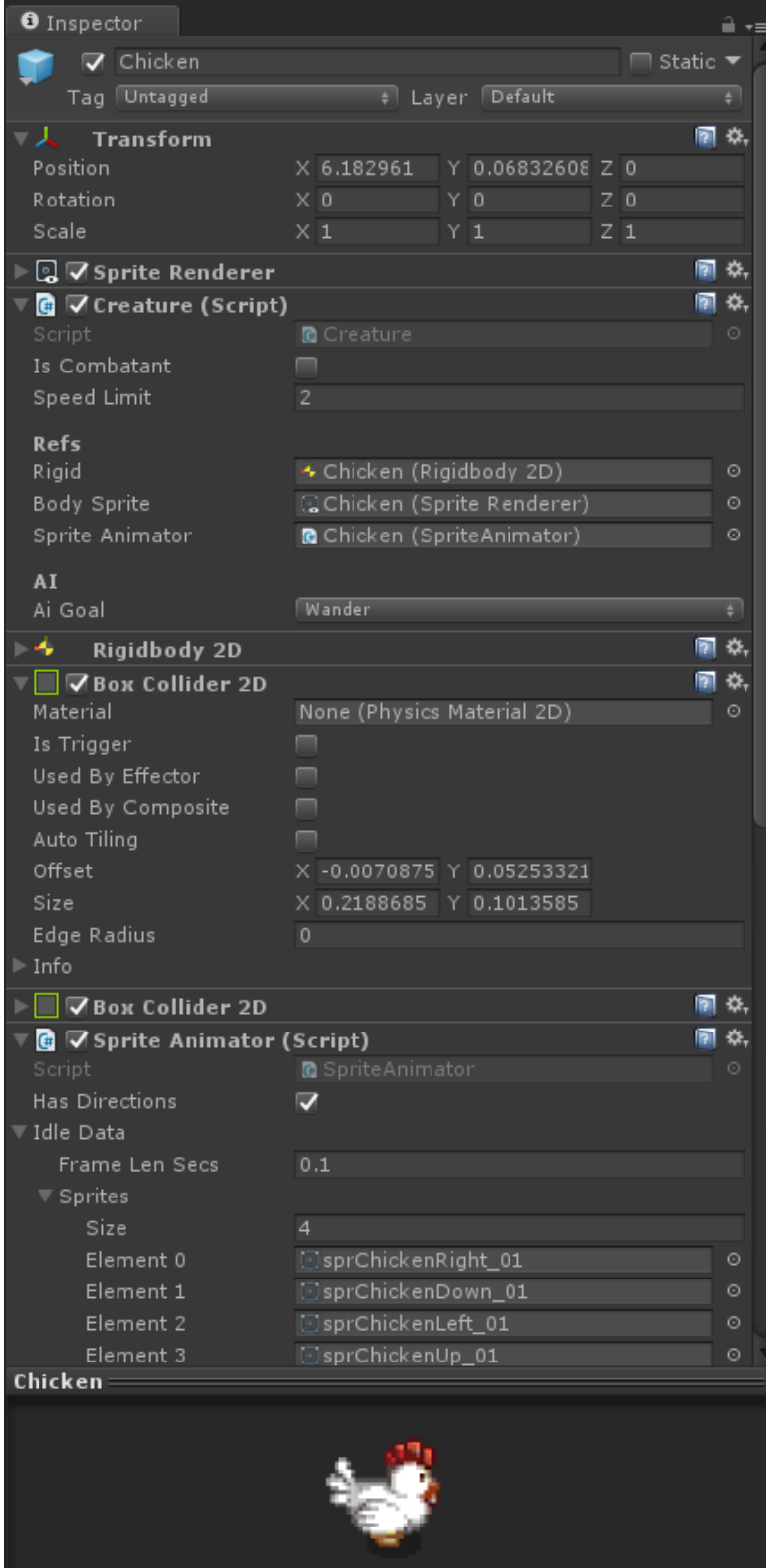
- Related to the physics engine are two sets of MonoBehaviour methods which are called automatically when objects with Colliders or Triggers interact:

OnCollisionEnter(Collision coll) OnCollisionExit(Collision coll) OnCollisionStay(Collision coll)	These happen for Collider-to-Collider collisions
OnTriggerEnter(Collider other) OnTriggerExit(Collider other) OnTriggerStay(Collider other)	These happen for Trigger-to-Trigger interactions

- The **Collision** argument provides information on the game object that has collided with us, as well as additional information about the collision itself (e.g. speed of impact – see section 4 of notes)
- The **Collider** argument is simply a reference to the trigger component that interacted with ours
- The Unity manual gives further detail on these, e.g.:
  - https://docs.unity3d.com/ScriptReference/Collider.OnCollisionEnter.html

Object interactions with Colliders and Triggers

- 'OnCollision' refers to an actual physics interaction between two Colliders (e.g. BoxCollider or SphereCollider components)
- 'OnTrigger' is the same thing except here the Collider has its Trigger flag set to true.
- Triggers enable game-logic interaction when you don't want physical collision responses.



Lab 3: Asteroid assault on Mars and its moons!

- This builds on the work from the last 2 weeks
- We already have Mars rotating, with Phobos and Deimos orbiting around it
- We also have keyboard control of the camera
- This week, we're instantiating asteroids at run-time, and setting them moving (using physics)
- Asteroids which collide with anything will be destroyed
- Asteroids which pass offscreen will also be destroyed

Triggers (sensors)

- Invisible components that are activated (triggered) when a character or other object passes inside them
- Very common (and useful) mechanism in games
- In Unity, use a Collider with its Trigger property checked
- Various things they're used for in Goblins & Grottos (a game I wrote using Javascript/PixiJs):
  - Goblin speech
  - Cutscene triggers
  - Trap triggers
  - Level exit (finished)
  - Etc.

Coroutines

- https://unitygem.wordpress.com/coroutines/
- Coroutines are a very useful feature provided by the MonoBehaviour class
- Why so useful? Because the code related to something occurring in the game is kept all in one place, even if that something takes a long time to occur, or occurs in stages
- Coroutines are special functions that can pause their execution
- They're very useful for delayed execution, for making things happen in steps (with a defined delay between each step), or for waiting for something else to complete before executing
- They're not threads, in that they run in the same thread as the main Unity process.. but they're much simpler to use, and in any case most of the Unity SDK is not thread-safe
- Coroutines are started with a call to **StartCoroutine( )**, and they must have the **IEnumerator** return type
- Technically, the way these work is that Unity maintains a list of active Coroutines and on each game loop it wakes up any that are now due to wake up

Coroutines

- When the coroutine is activated it will execute right up to the next **yield** statement and then it will pause until it is resumed
- Various values can be supplied with the yield statement.. E.g. **yield return null** simply waits until the next frame

```
private void Start() {
    Debug.Log("Start method");
    StartCoroutine(TestCoroutine());
    Debug.Log("Start method ends");
}

private IEnumerator TestCoroutine() {
    Debug.Log("TestCoroutine");
    while(true)
    {
        Debug.Log("Here");
        yield return null;
        Debug.Log("There");
    }
}

private void Update() {
    Debug.Log("Update");
}
```

This coroutine has a loop that runs as long as the calling object is active

Start method
TestCoroutine
Here
Start method ends
Update
There
Here
Update
There
Here
.
.

The most useful (but not all) Coroutine yield values

- **yield return null;** – the coroutine is continued the next time that it is eligible, normally on the next frame
- **yield return new WaitForSeconds(3f);** – causes the coroutine to pause for a specified time period (3 seconds, in this example)
  - *Be aware of the garbage implications of this*
- **yield return StartCoroutine(OtherCoroutine());** – waits until the other coroutine has run to completion before the yielder is resumed

- **yield break;** - stops the coroutine and exits (i.e. this is the equivalent of a **return** statement in a normal function)

Example
(Part of Gun.cs in DemonPit)

```
private IEnumerator ParticleShootingSounds() {
    AudioSource asrc = GetAudioSource();
    asrc.clip = shootSounds[0]; // shooting intro
    asrc.volume = GameManager.instance.optSoundsVolume;
    asrc.Play();

    while (isParticleShooting) {
        if (!asrc.isPlaying) {
            asrc.clip = shootSounds[1]; // shooting loop
            asrc.loop = true;
            asrc.Play();
        }
        yield return null;
    }

    // fade out
    while (asrc.volume>0f) {
        yield return null;
        asrc.volume -= 1f/15f; // fades over 0.25 secs
    }
    asrc.Stop();
}
```

Pseudocode Example (Part of Gun.cs in DemonPit)

```
ShootingAnimations coroutine:
    elapsedTime = 0
    startTime = Time.time
    While (elapsedTime < recoilKickbackTime)
        yield return null
        elapsedTime = Time.time - startTime
        Animate Recoil Position for elapsed time
    End While

    elapsedTime = 0
    startTime = Time.time
    While (elapsedTime < recoilRecoveryTime)
        yield return null
        elapsedTime = Time.time - startTime
        Animate Recoil Position for elapsed time
    End While
End Coroutine
```

Invoke and InvokeRepeating

- This provides a simpler way of running a complete method some time in the future, either as a one-off or a repeating call

```
public class ExampleScript : MonoBehaviour
{
    // Launches a projectile after 2 seconds

    Rigidbody projectile;

    void Start() {
        Invoke("LaunchProjectile", 2f);
    }

    void LaunchProjectile()    {
        Rigidbody instance = Instantiate(projectile);
        instance.velocity = Random.insideUnitSphere * 5f;
    }
}
```

- To invoke repeatedly:

```
InvokeRepeating("LaunchProjectile", 2.0f, 0.3f);
```

Be Aware of Changes During Yield Time

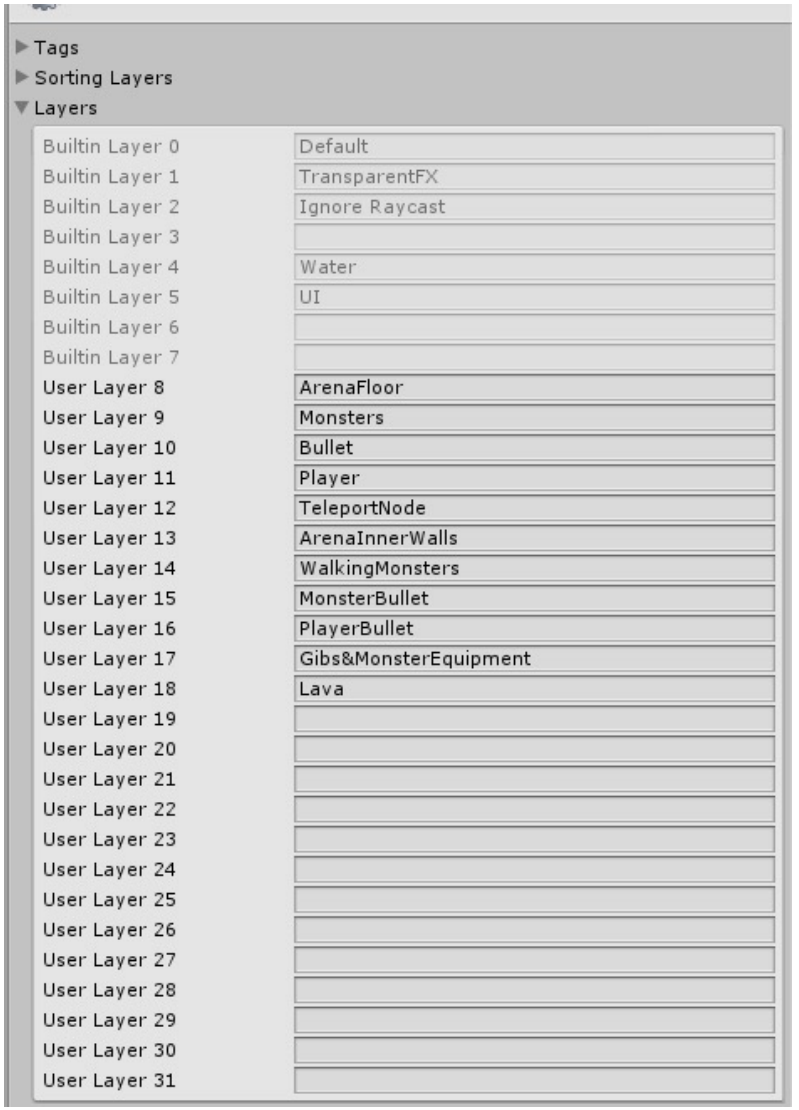
- You always have to be careful with asynchronous programming: something important may have changed during the 'down time' of a coroutine
- E.g. in DemonPit, the monsters flash a yellow colour when they're hurt but not killed
- The yellow flash takes some time and is performed using a coroutine which gradually modifies some settings to make the yellow effect fade in and then out again
- After each yield command in the coroutine, it is necessary to check whether the monster has been killed during the pause
- If the monster has been killed before the coroutine is complete, the coroutine exits immediately (in this case, a different coroutine will now be controlling the death animation)

Layers

- https://docs.unity3d.com/560/Documentation/Manual/Layers.html

- Layers let you group game objects into categories
- You can then decide which layers interact with each other in the physics simulation
- Or you can use an object's Layer to decide what happens when it collides with another object
- You can also apply raycasts (see later) on specific objects only.. E.g. to determine whether a monster can see a player, you would cast a ray between the position of the monster's head and the player's head, but only let the raycast consider walls/floors - i.e. it would ignore any creatures in the way
- Edit > Project Settings > Tags and Layers

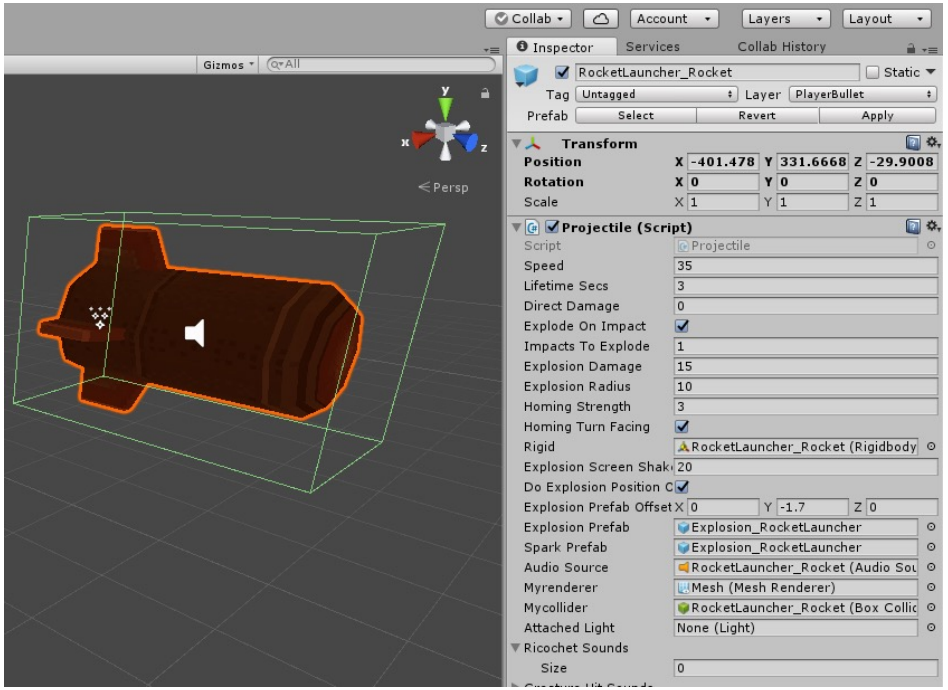
- Don't confuse Layers with Sorting Layers: the latter is for deciding the draw order of Sprites

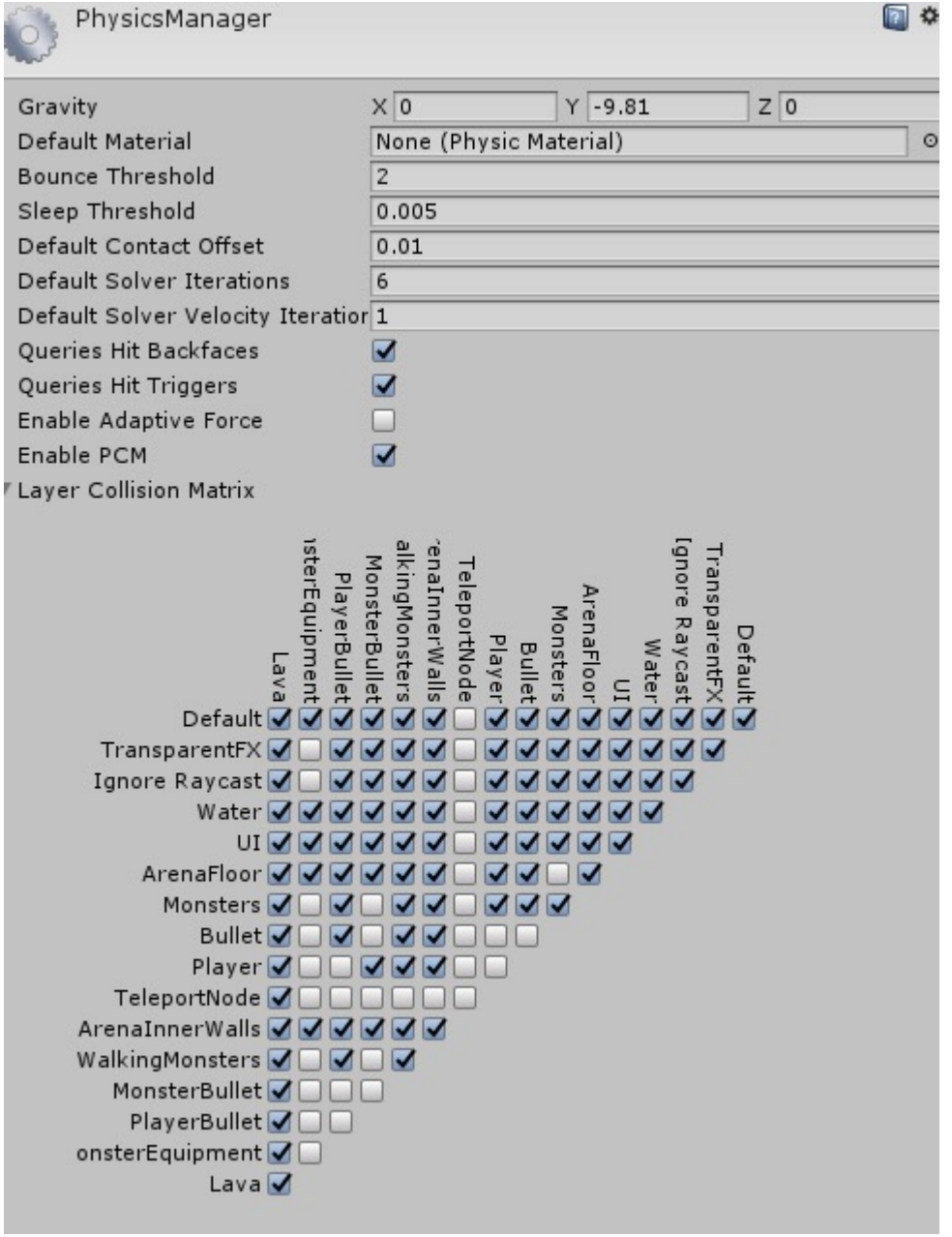


Layers and Physics Interactions

- Edit > Project Settings > Physics

- Here's an object whose layer is set to PlayerBullet:





How do we know what has collided with us in OnCollisionEnter?

- Layers
  - Use GameObject.layer to determine the layer of a game object - *as an integer*
  - LayerMask.NameToLayer("MyLayer") will give you a corresponding integer for a named layer
- Tags
  - Similar to layers, you define a set of tags for the game (for grouping objects by category) and then each prefab has a specific tag chosen
  - Use gameObject.tag to find the tag (as a string) of an object
  - This is simple, *but creates garbage*.. which is bad if done often (see later for optimisation and garbage control tips)
  - Better to use gameObject.CompareTag("Tag")
- GetComponent<>()
  - You could check for existence of a specific script on a GameObject, to decide what type of thing it is (GetComponent<>() returns null if the component isn't found)

DemonPit: Pickupable::OnTriggerEnter()

- The Pickupable script is attached to the various things in DemonPit that the player can pick up by running into: guns, ammunition, and health packs.
- Each of these prefabs has a trigger attached to it, and the Pickupable script includes:

```
void OnTriggerEnter(Collider other) {
    if (other.gameObject.layer==GameManager.layerPlayer) {
        Player.PickedUpItem(this.itemType);
        PoolManager.ReturnGameObject(this.gameObject);
    }
}
```

(Optional) Example

- This is a nice examples but feel free to skip it if the additional code (related to animating a 2D sprite) is confusing

E.g. a 'BumpToAnimate' script

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BumpToAnimate : MonoBehaviour {
    // this object cycles once thru its
frames when it is bumped into by a collider

    // inspector values
    public float frameLenSecs = 0.1f;
    public Sprite[] sprites;
    //

    private int currFrame = 0;
    private SpriteRenderer spriterenderer;
    private bool isAnimating = false;
```

- **SpriteRenderer is the Unity component used to render 2D sprite objects (normally used with an orthographic camera).. we'll discuss 2D games in detail later**
- **See next slide!**

```
public IEnumerator Animate() {
        spriterenderer = GetComponent<SpriteRenderer>();
        spriterenderer.enabled = true;

        currFrame = 0;
        spriterenderer.sprite = sprites[currFrame];
        isAnimating = true;


        while (true) {
            yield return new WaitForSeconds(frameLenSecs);

            currFrame++;
            if (currFrame>=sprites.Length) {
                spriterenderer.sprite = sprites[0];
                isAnimating = false;
                yield break;
            }

            spriterenderer.sprite = sprites[currFrame];

        }
    }

    void OnCollisionEnter2D(Collision2D coll) {
        if (!isAnimating && !coll.otherCollider.isTrigger)
            StartCoroutine( Animate() );
    }

    void OnTriggerEnter2D(Collider2D other) {
        if (!isAnimating && !other.isTrigger)
            StartCoroutine( Animate() );
    }
}
```

'BumpToAnimate'

- This shows the inspector settings for this bush object's BumpToAnimate component

- Note that the bush and chicken prefabs both have two BoxColliders..
- The smaller one is a normal collider which covers just the trunk of the bush or feet of the chicken
- The larger one surrounds the whole sprite and is a trigger (rather than a physical collider)
- A creature can walk behind the bush which will 'rustle' it without blocking the creature from walking.